2

CMU-CS-78-101

2

MODULARIZATION AND HIERARCHY IN A FAMILY OF OPERATING SYSTEMS [1]

A. N. Habermann Peter Feiler

Lawrence Flon Loretta Guarino

Lee Cooprider Bob Schwanke

Carnegie-Mellon University

February 1978

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

Approved for public release;
distribution unlimited.

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DDC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(14) CMU-CS-78-1Ø1

(19) REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. ~~REPORT NUMBER~~ (18) AFOSR-TR-78-1Ø89

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle) (6) MODULARIZATION AND HIERARCHY IN A FAMILY OF OPERATING SYSTEMS.

5. TYPE OF REPORT ~~& PERIOD COVERED~~ (9) Interim rept.,

6. ~~PERFORMING ORG. REPORT NUMBER~~

7. AUTHOR(s) (10) A. N. Habermann, Lawrence Flon, Lee Cooprider, Peter Feiler Loretta Guarino [redacted]

8. CONTRACT OR GRANT NUMBER(s) (15) F4462Ø-73-C-ØØ74, ARPA Order-2466

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Carnegie-Mellon University
Department of Computer Science
Pittsburgh, PA 15213

10. ~~PROGRAM ELEMENT, PROJECT~~, TASK AREA & WORK UNIT NUMBERS
61101E
A02466/7 (16) 2466

11. CONTROLLING OFFICE NAME AND ADDRESS
Defense Advanced Research Projects Agency
1400 Wilson Blvd.
Arlington, VA 22209

12. REPORT DATE (11) Feb[redacted]78 (17) Ø7

13. ~~NUMBER OF PAGES~~
24

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
Air Force Office of Scientific Research (NM)
Bolling AFB, DC 20332

15. SECURITY CLASS. ~~(of this report)~~ (12) 25p.
UNCLASSIFIED

15a. DECLASSIFICATION/~~DOWNGRADING~~ SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The objective of the "Family of Operating Systems" project has been to investigate the feasibility of constructing systems which use identical or similar resources and share basic design decisions. The concepts of "module", "address space" and "hierarchy" have been used with special care.

Common to all family members is the virtual memory facility which controls dynamic address space transitions. Family members may differ in the facilities they provide in static address spaces.

continued

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

next page 4Ø3 Ø81 Gou

20. abstract (continued)

This report presents an overall description of the FAMOS system. Section 1 describes the basic ideas underlying the FAMOS system and Section 2 describes the implementation. A more detailed description is found in the official documentation of the FAMOS system.[2] This documentation consists of a number of "module documents". Each module document comprises two parts, an introductory description which specifies the function and dependency of a module and a "type description" which defines the representation and implementation of a module as static address space.

CMU-CS-78-101

MODULARIZATION AND HIERARCHY IN A FAMILY OF OPERATING SYSTEMS [1]

A. N. Habermann Peter Feiler

Lawrence Flon Loretta Guarino

Lee Cooprider Bob Schwanke

Carnegie-Mellon University

February 1978

The objective of the "Family of Operating Systems" project has been to investigate the feasibility of constructing systems which use identical or similar resources and share basic ,design decisions. The concepts of "module", "address space" and "hierarchy" have been used with special care.

Common to all family members is the virtual memory facility which controls dynamic address space transitions. Family members may differ in the facilities they provide in static address spaces.

This report presents an overall description of the FAMOS system. Section 1 describes the basic ideas underlying the FAMOS system and Section 2 describes the implementation. A more detailed description is found in the official documentation of the FAMOS system.[2] This documentation consists of a number of "module documents". Each module document comprises two parts, an introductory description which specifies the function and dependency of a module and a "type description" which defines the representation and implementation of a module as static address space.

Key Words and Phrases: incremental machine design, module, data type, address space, virtual memory

[1] This work was supported in part by the Defense Advance Research Projects Agency under contract F44620-73-C-0074, and in part by the National Science Foundation under grant DCR74-24573.

[2] The documentation is available upon request.

1. The Family Concept

1.1. Introduction

The design, specification, implementation, documentation, and maintenance of a general purpose operating system is without question a huge project, requiring many man-years of effort. The finished product is usually just that - general purpose. Such a system can (is designed to) behave as any or all of a batch, timesharing, communications, process control system, etc., at any time. A general purpose system cannot be as efficient in any of its roles as would be a system specifically designed for one particular purpose. Unfortunately, the development cost of even a uni-purpose system usually precludes the construction of several independent such systems.

The FAMOS project had two major goals. The first was a demonstration of the feasibility of designing a *system family*. The idea of a family of operating systems derives from [Parnas 72a] and [Price 73]. Members of a system family are developed as far as possible along common lines to avoid as much re-design and recoding as possible. The software system family concept is somewhat analogous to the hardware concept illustrated by the IBM System/360 series or the DEC PDP-11 family, although hardware families are generally oriented towards very similar user interfaces among all family members. It might be argued that a particular computer is not well-suited for more than one or two types of service, and therefore does not merit the development of differing systems. While this may be true for larger machines (though the manufacturers might disagree), it is definitely not true for the proliferation of minicomputers on the market, most of which have a very large range of possible applications, and tend to cost less than the systems which run on them.

Our second goal concerned the documentation and description of the family. In [Habermann 73], we proposed the description of a system at various levels of abstraction. Each partial description consists of a specification, a list of decisions, an analysis, and an implementation description. The feasibility of this method has been tested by applying it to the family design. An important aspect of the description is that the specification of a system facility is separated from its implementation. As a result, the implementation may be changed without affecting programs that rely solely on the specifications.

1.2. Design Methodology

The approach used in structuring the design of the family is *incremental machine design*, similar to that introduced by Dijkstra in the T.H.E. system [Dijkstra 68] and used in [Liskov 72] and [Neumann 74] among others. Incremental machine design denotes building a system up level by level. Each level defines a *virtual machine* for use by higher levels. This machine models a hardware machine closely in the facilities it provides to its users.

The various system concepts are introduced in such an order that decisions which restrict the family are postponed as far as possible. The decisions which are made are only those of specification, allowing various family members to share the specification of a level without necessarily sharing its implementation.

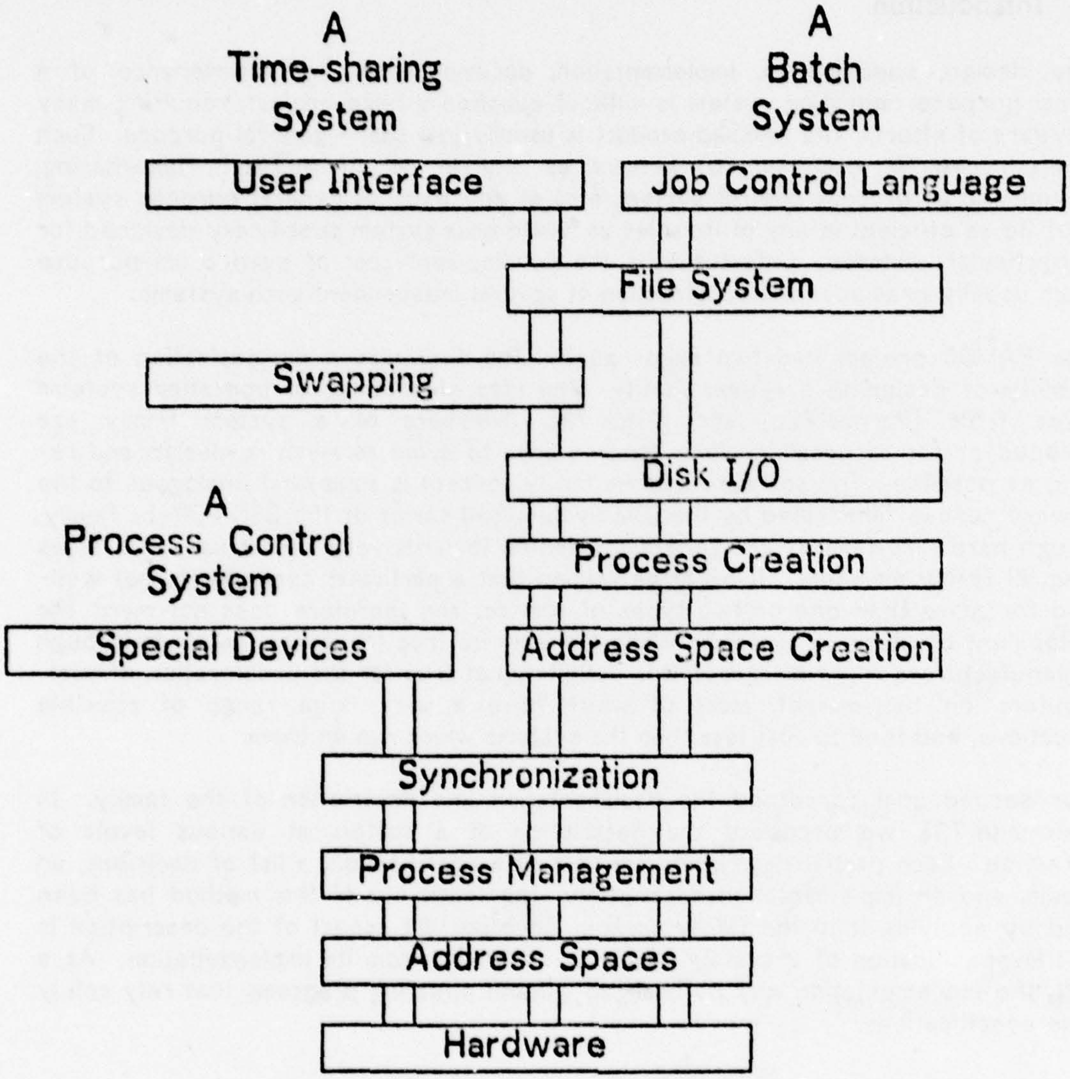


Figure 1: A family of operating systems

Figure 1 shows a possible relationship between family members. All the systems share the specification of the address space management, process management, and synchronization levels. The process control system, however, will not need to create processes dynamically, nor to provide user facilities like the other systems. Instead, it defines the special devices it will control. The batch system may use fixed memory partitions, and not need to do any of the swapping, which a timesharing system must do. A very useful result of the design should be the compatibility of such things as file systems between both the timesharing and batch systems supplied by a vendor.

1.3. Functional Hierarchy

The fact that we use the concept of partitioning a system design into levels implies nothing *per se* about the interaction among those levels. In particular, the hierarchical structuring is based upon functions - not processes as employed in the T.H.E. system. Each level is comprised of a set of functions whose names are statically known. The levels L_0, L_1, . . . L_n are ordered such that functions defined in level L_i are also known to L_{i+1} (and, at the discretion of L_{i+1}, to L_{i+2}, etc.). L_0 corresponds to the hardware instructions of the target machine. Each level, in fact, is regarded as providing new "hardware" to the next higher level.

It is most important that the hierarchy is among functions. One of the arguments against the T.H.E. design is the overhead associated with inter-level communication among processes. In a functional hierarchy where functions may actually be macros, a sequence of function calls may result in a single machine instruction (or possibly none at all) when the system is compiled. It is the system design which is hierarchical, not its implementation.

1.4. *Modularization and the Grain of Hierarchy*

Parnas [Parnas 74] has noted that the notions of *level* and *module* do not necessarily coincide. We wish to elaborate upon this somewhat.

Information modules [Parnas 72b] are comprised of some data structures (possibly) and a set of functions which share knowledge of a particular design decision (reflected, for example, in the details of the data structures). A level is a set of function names which are implemented via functions in lower levels. There exists no necessary relationship between the two concepts. This not only allows the division of a single level into several distinct modules, but in addition allows for the selective spanning of several levels by a single module! For example, a process manager may be implemented above the memory manager so that it can create processes dynamically. However, the memory manager may need scheduling facilities in order to suspend allocation requests when memory is full. This apparent demonstration of the futility of the level hierarchy can be resolved by the division of a module into more than one level.

In figure 2 the memory and process management modules are interleaved in such a way as to not violate the functional hierarchy. The two pieces of the memory manager are part of the same module because they share knowledge about how virtual memory structures are implemented (e.g. segment tables and descriptors). Imposing a functional hierarchy may therefore (and in some cases does) result in a proliferation of levels, which produces a finer grained hierarchy than those found in systems previously developed. However, as mentioned earlier, this does not necessarily add to run-time expense.

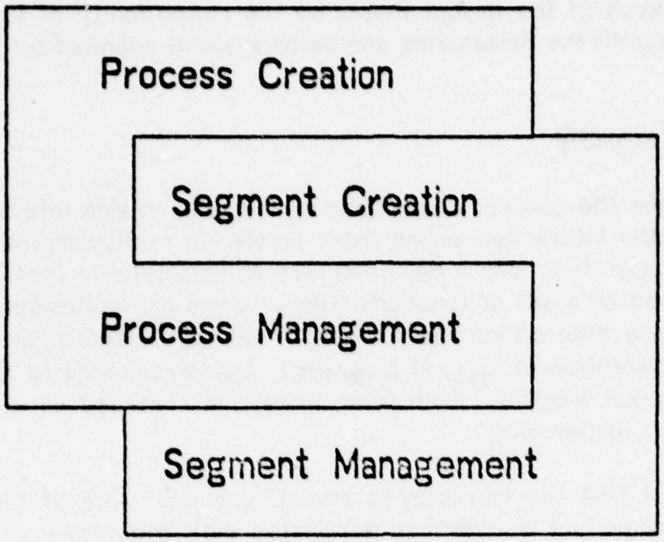


Figure 2: Modules and hierarchy

1.5. Virtual Machine Definition

A good example of a firm boundary between levels is the boundary between a hardware machine and its programs. In this case the hardware provides a set of data registers (memory, device control words, status words, accumulators, etc.) and a set of instructions for manipulating those registers. A program written for this particular machine is considered to be at a higher level, and *it may or may not use the hardware "correctly"*. There is no opportunity for violation of the order of the program and the hardware.

The hardware can be used in a variety of ways, some of which have been anticipated, and others which are erroneous. A typical example of erroneous use of hardware is an attempt to branch to an invalid address. Nevertheless, the hardware is considered to be correct if the following type of statements hold:

1) Valid instructions operating upon valid registers yield results as predicted by the specifications (e.g. the add instruction on large numbers results in overflow).

2) No sequence of instructions can cause irreparable damage to the machine.

3) When invalid instructions or invalid operands to instructions are detected, a specified action is taken (such as a trap) and side effects on registers behave as specified for each condition.

Note that the hardware has not failed if the programmer places the address of his program code into the stack pointer register, or fails to provide a valid interrupt word before the clock expires. The correctness of the hardware level is determined without regard to its use. The software level however, is dependent upon the correct operation of the hardware (although it may attempt to be tolerant of intermittent errors or failures in a restricted portion of the hardware).

The hardware analogy provides a prototype for virtual machine interfaces. A virtual machine is a programmable computer with registers, instructions, and specified actions for all improper uses of the machine. In general, a virtual machine is an incremental modification of a lower level machine called its *base*. Using the term "facilities" to mean the registers, instructions, and asynchronous activities of a given machine, the possible modifications which a virtual machine can apply to its base may be classified as:

1) the hiding of a subset of the facilities - i.e. making them unavailable to higher levels.

2) the definition of new facilities.

3) the systematic modification of a subset of the existing facilities.

Some of the new registers may serve the function of trap or interrupt words for higher level programs. (We differentiate traps, which result directly from program actions, from interrupts, which result from external asynchronous events.) A trap word provides an address to which control is to be passed if the trap or interrupt condition occurs. The side effects of a trap are part of the specifications of the virtual machine. Using this mechanism, the higher level program can exercise the functions of the machine, handle erroneous uses (which in some cases may be desirable - e.g. page faults), and process the results of asynchronously operating aspects of the virtual machine.

1.5.1 Example 1

A low level machine might have floating point operations which can result in underflow in magnitude, causing a trap through a special register known as the floating point underflow trap word. A possible higher level machine might make a small change to the base machine so that

1) the floating point underflow trap word is hidden.

2) A "floating point underflow count" register and two operations on it, read and clear, are defined.

3) The floating point instructions of the base machine are systematically modified so that the phrase "causes a trap through the floating point underflow trap word" in the documentation is changed to read "causes the floating point underflow count word to be incremented".

A straightforward implementation of this machine would be created by placing the address of the underflow count increment routine in the floating point underflow trap word, and making available two routines "clear underflow count" and "read underflow count". This · software implementation requires that several other base machine registers, namely the memory locations occupied by the count word and routines must also be hidden.

1.5.2 Example 2

The function of a virtual machine might be to provide multiple versions of a facility provided only once in the base machine. For example, a virtual clock level could replace the single clock of the hardware with several clocks which may be started and stopped independently of one another. In this case, many new interrupt conditions would be created, and an interrupt register would be provided for each of the clocks defined by the virtual clock level.

1.6. Implementation Alternatives

Should the rules implied by the level structure and the modular structure be checked at compile time or at run time? We decided to check functional hierarchy at compile time and module boundaries at run time. A justification of these decisions follows.

A compile time check has the obvious advantage that the check is performed only once, before execution starts. Moreover, a compiler can optimize the code across level or module boundaries and gain a reduction in space and time requirements. It seems that for this reason both level hierarchy and modular structure should be checked at compile time. Regarding hierarchy, we can afford to check the validity of a function call at compile time because of the earlier design decision that function names are non-computable objects and can specify the level at the definition site. That is, function names behave as constants which are known at compile time. (This decision does not preclude that the parameters given in a function call, or their types, may trigger the invocation of a particular version of the function, but all these versions carry the same function name and are at the same level.) Since a level is defined as a set of function names, the rules defining hierarchy can be checked at compile time. Thus, a compiler can optimize code across level boundaries. As a result, it may be hard to find hierarchy in the compiled code, a situation not unlike nested control structures which are compiled into jump instructions.

Within modules, addresses of objects are computed at run time. This means that a program may generate an incorrect address and unintentionally modify arbitrary locations. In order to limit the damage which results from incorrect address computations, we associate a module with a collection of memory cells addressable only by the functions which belong to the module. (This corresponds very closely to the "invisible" registers accessible by arithmetic logic or microcode in hardware.) Since addresses are computed at run time, the check that a generated address is within the bounds of a module must be made at run time. Inter-module function calls require a change of environment which may or may not be expensive, depending upon the appropriateness of the hardware. However, data local to a module is completely protected from external addressing errors. Intra-module calls, since they do not require a change of environment, can be compile-time optimized even across levels.

1.7. Description and Documentation

Description and documentation form an integral part of the design task. The resulting family is not merely defined by its code, but exists as a document describing modules of the systems at various abstract levels. The data associated with a module is described by abstract data types. An abstract data type describes to the user of an object the abstract states of such objects and the functions which manipulate them. The "introductory description" of a module specifies the data types used in a module. A "type description" gives the data representation and the implementation of operations for each data type.

The description method given above facilitates the understanding and modification of modules. It allows a programmer to get acquainted with the system family without having to go through the tedious experience of deriving meaning from the code. In particular, by separating the specifications in the introductory description from the implementation in the type description, one can understand how to use a module without having to understand every detail of the implementation. In addition, the implementation of the abstract states of an object, or the implementation of the operations *defined in a type definition, can be changed without affecting the users of* the typed object, provided that the specifications remain unaltered. An introduction to the use of abstract data types as a design tool may be found in [Flon 75].

Dijkstra observed in [Dijkstra 68] that level hierarchy facilitates the debugging process. The hierarchy makes it possible to debug the levels one by one, starting at the lowest level. Abstract data types provide an additional debugging tool, since the operations on typed objects can be tested independently of their call sites. Moreover, if a bug is found, the programmer can be sure that the errant code is limited to the type definition in which the bug occurs. The strict separation of specification and implementation makes it impossible for an implementation change in one place to require additional changes in an arbitrary number of other places.

2. Family Implementation

2.1. Introduction

The three methods of transition among the various modules of a system can be summarized as:

1) simple intra-module function calls

2) inter-module function calls

3) virtual traps and interrupts

The concepts involved in this statement are considered to be universally basic to the family, more so than any others. Therefore, a protected version of these facilities comprises the lowest system level. Price [Price 73] has shown that such a basis is sufficient to guarantee adequate protection for the system and its users. His prototype implementation established that inter-module calls would not be overly expensive on a machine with appropriate hardware of a simple nature. The implementation described in this paper is a follow-up to Price's work. The design is somewhat simplified with respect to inter-module connections. An important extension is the processing of virtual interrupts, mentioned previously. The integration of this level with the design of higher levels has led to more extensions and a growth of confidence in the utility of the features provided.

2.2. The Address Space

A module is characterized by

1) the information for which it is responsible

2) the set of functions it provides to other modules for manipulating that information

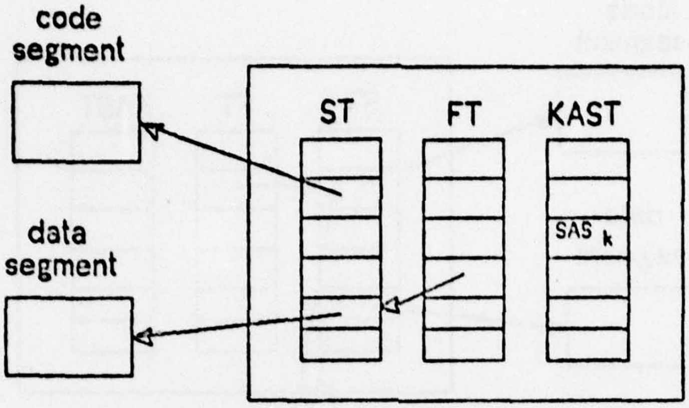
3) the set of modules of which it knows the existence

The instantiation (execution) of a module is characterized by

1) the function invoked and the subset of information it needs to operate

2) any additional information passed to it via the parameter mechanism

The concept of *address space* is introduced to implement these notions. When a module is defined, a *static address space* (SAS) is created for it (figure 3). Contained in the SAS are a *segment table* (ST) which represents information local to all instances of the module, and a *function table* (FT) which is a vector of information



⊕iFigure 3: Static address space]

about the invocation of each of the possible module entry points. The ST is a vector of *segment descriptors*, each of which identifies a segment of memory, either by means of a physical address or indirectly via a reference to a segment in another SAS. The latter case allows for the sharing of segments among address spaces. Also provided in the SAS is the *known address space table* (KAST) which consists of a vector of SAS names.

The functions in a module have no direct access whatsoever to any of these tables, since an instance of a function runs in virtual memory i.e. all addresses are relocated. Instead, an active address space manipulates these tables by operations provided by the virtual memory level (VM), including *ASCALL* and *ASRETURN*, which comprise the mechanism for inter-module calls.

ASCALL takes as parameters

1) a KAST index (i)

2) an FT index (j)

3) a list of parameters in the form of ST and PST indices (k,l,m,. . .)

4) an optional ST or PST index q for the return segment.

In effect, an inter-module call can be read as "call the i'th module I know about, invoking the j'th function it provides. Pass as parameters to that module the k'th, l'th, m'th, etc., segments I know about." If the function returns a segment, make it the q-th segment.

The result of an *ASCALL* is the creation of a *dynamic address space* (DAS). A DAS consists of a *working set* (WS), a *parameter segment table* (PST), and an SAS reference (figure 4). The WS is a vector of segment descriptors which comprise the virtual address mapping for this DAS. The WS is initialized by VM according to information

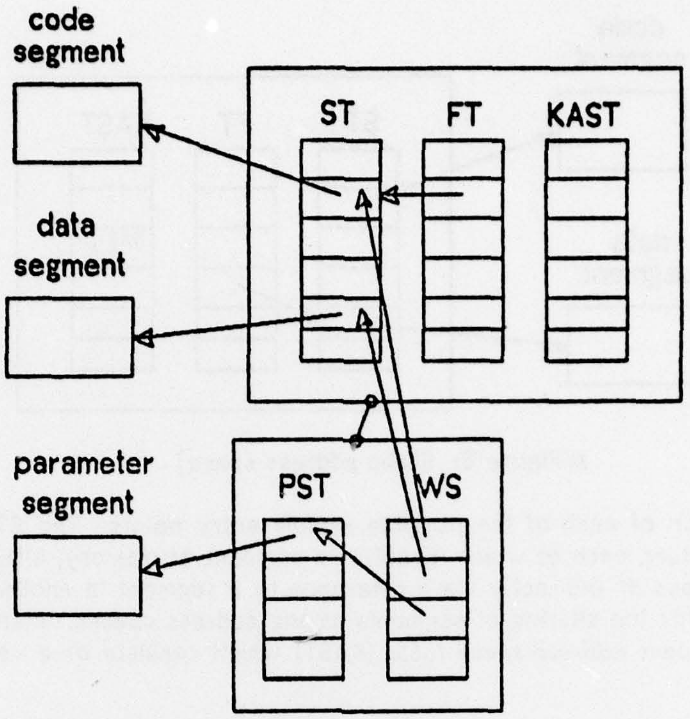


Figure 4: Dynamic address space

taken from the called-address-space's FT, i.e. the code segment and a local data segment. A virtual address of the form (w,d) indicates the w'th WS segment with displacement d. The PST is initialized by VM with indirect segment descriptors which represent the parameter segments passed by the calling-address-space (access control may be restricted).

When a function executes an *ASRETURN* instruction, the return segment (if any) is stored in the caller's address space, the function's DAS is erased and the DAS which called it is resumed following its *ASCALL*. During execution, a function may load and unload the WS and ST with segments from ST or PST via the instructions *SEGLOAD* and *SEGUNLOAD*.

<u>Example</u>: The behavior of *ASCALL* is illustrated in Figure 5. The calling program would like to open a file. It makes an *ASCALL* on the file manager, passing as a parameter a segment containing the file name. A segment will be returned which is an open file.

The program's static address space (SAS_{user}) contains a segment table which has descriptors for at least a code segment, data segment and file name segment. The known address space table of SAS_{user} contains a reference to the file manager address space (SAS_{file}). The file manager ST contains descriptors for (at least) a code and data segment. The data segment might contain the directory structure, mapping file names to physical files. When the user program is executing, it has a dynamic

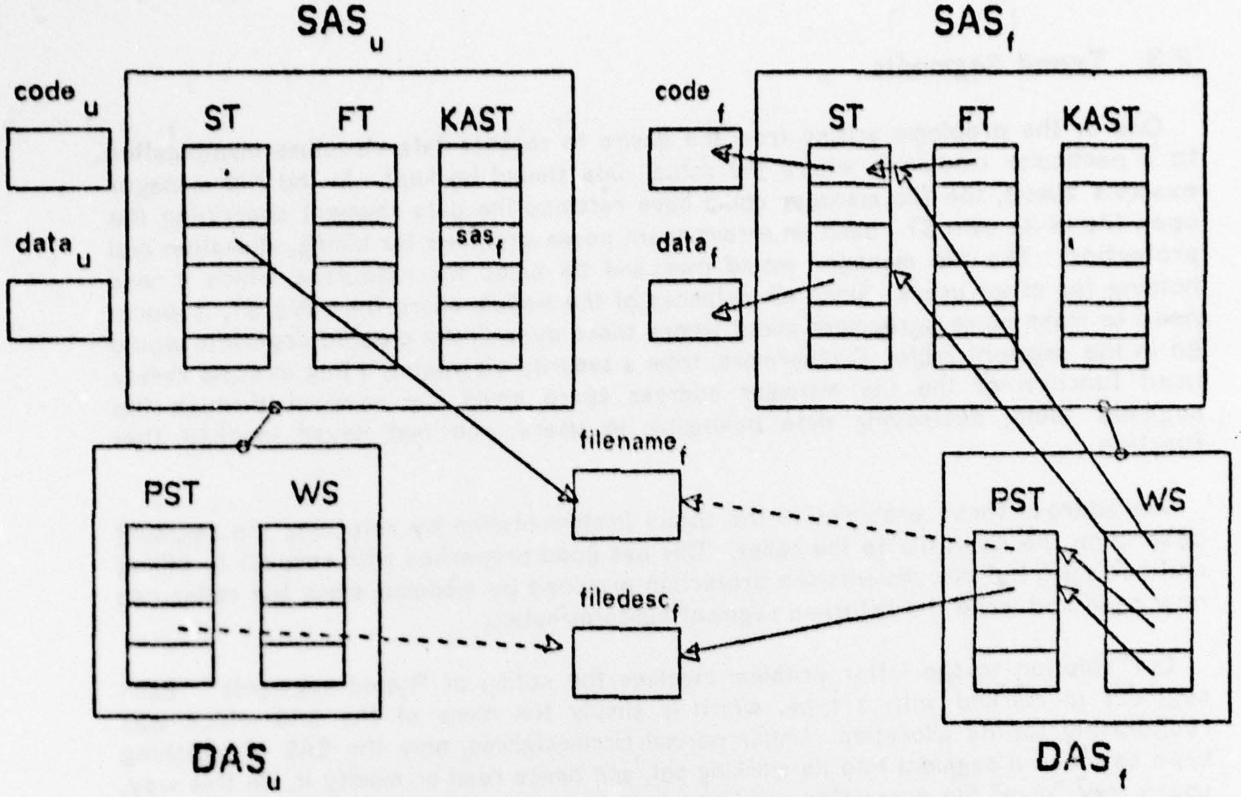


Figure 5: ASCALL to OPEN a file

address space (DAS$_{user}$) whose working set entries contain the ST and PST indices of the segments currently addressable. DAS$_{user}$ also has a reference to SAS$_{user}$ and a parameter segment table.

When the program executes

ASCALL(kast[3], 1, st[3], pst[3])

a new DAS (DAS$_{file}$) is created for the file manager (kast [3] = SAS$_{file}$). The function to be executed is the first in the FT of SAS$_{file}$. The function table descriptor in the FT indicates that the code for the function is to be found in the first slot of ST. The new PST is loaded with a descriptor for the third segment in the SAS$_{user}$ segment table (the data segment containing the file name). The last parameter indicates that a segment will be returned, whose descriptor should be stored in the third slot of PST$_{user}$. DAS$_{file}$ then executes the code segment, loading its data segment and the parameter data segment into its WS, and eventually creating a segment to describe the state of the open file. When the called function completes and does an ASRETURN, the descriptor for the return segment is stored in the caller's PST (as specified by the ASCALL), the file DAS is erased, and the user DAS is resumed.

2.3. Typed Segments

One of the problems arising from the desire to restrict data structure manipulation to a particular module is where the actual data should be kept. In the file manager example above, the file manager could have retained the data segment describing the open file in its own ST. Such an arrangement poses problems for billing, allocation and protection. The file manager would own and be billed for resources which it was holding for other users. Since all instances of the module share the same ST, it would need to make some agreement about where these dynamically created segments would go in the segment table. Furthermore, from a security viewpoint, a bug in some rarely used function of the file manager address space could run rampant through the segment table, destroying data belonging to users who had never invoked that function.

We address these problems in the above implementation by returning the segment describing the open file to the caller. This has good properties with respect to billing and security, but circumvents the protection provided by modules, since the caller can now read and write the returned segment indiscriminately.

Our solution to the latter problem involves the notion of "typed segments". Each segment is marked with a type, which is simply the name of the SAS which was responsible for its allocation. Under normal circumstances, only the SAS of matching type can load a segment into its working set, and hence read or modify it. In this way, users may "own" file descriptor segments, but they are prevented from changing them except via operations provided by the file manager address space. In certain circumstances, an address space may wish to grant another address space the right to load one of its segments, such as when two address spaces are communicating via a buffer. To handle such situations, we permit an address space to grant, to a called function, loading privileges on a parameter segment of the caller's type.

Example: After opening a file, the user program may wish to read the file. It does so by calling the READ function of the file manager (see figure 6), passing as parameters the file descriptor segment (of type "file manager") and a buffer segment (of the user's type), with loading privileges. This permits the file manager to write into the buffer segment the data it obtains from the file referred to by the file descriptor segment. Upon completion of the call, the data is available for the user, who retains ownership of both the buffer and file descriptor segments.

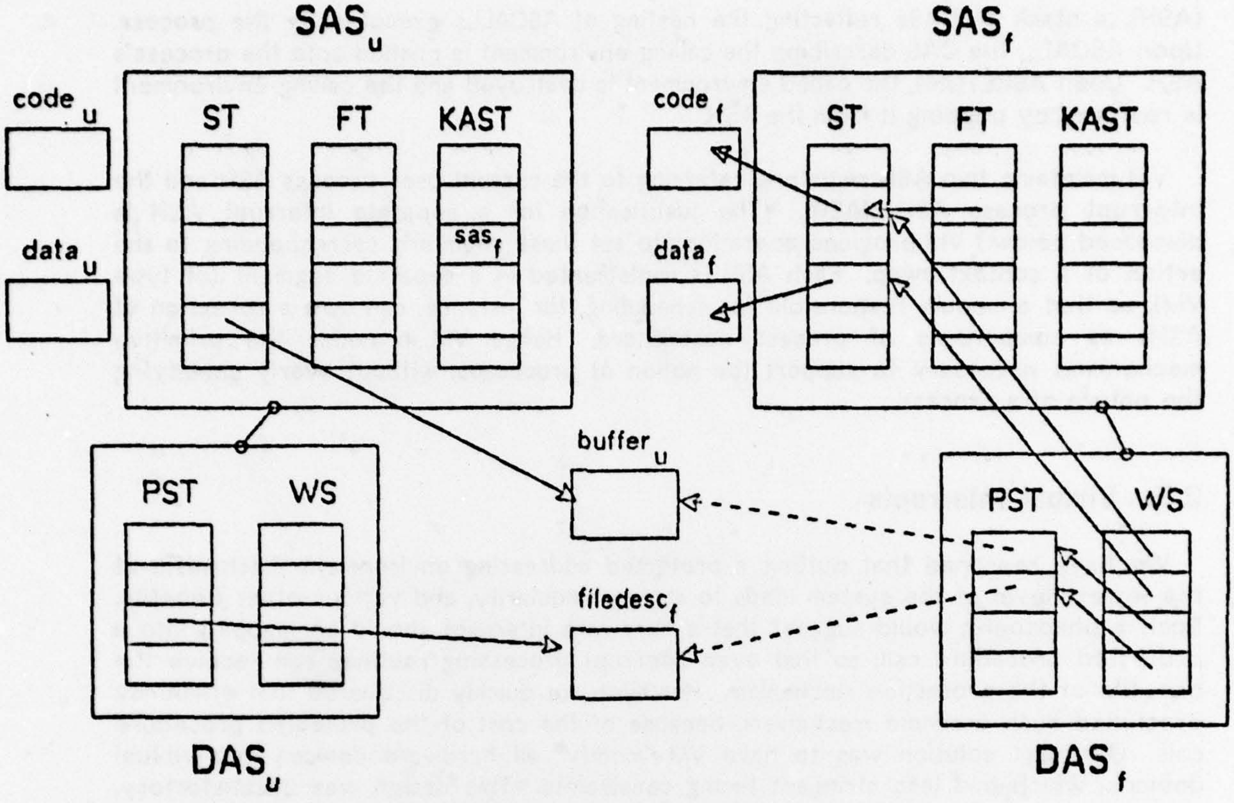


Figure 6: ASCALL to READ a file

2.4. Processes and Modules

Although the concept of process is unknown to the VM level, some thought must be given to the relationship between processes and modules in order that the higher levels have appropriate "virtual hardware" with which to work. In a conventional system, system code is viewed as being executed by separate processes. When a user requests a system function, his process is suspended while the system carries out his request. The protection needs which prompted this approach are provided in our system by the module mechanisms. Therefore a process can be thought of as a flow of control which passes among various modules, some of which are user-written and some of which are part of the system. A request for I/O, for example, involves the process actually executing system code.

This leads to a view of processes flowing between modules in a call/return manner, and allows for more than one process to execute a given module at the same time. Any necessary synchronization is defined in the module itself. The call/return discipline of control flow in a process is mirrored in VM by an address space history

(ASH), a stack of DASs reflecting the nesting of ASCALLs executed by the process. Upon ASCALL, the DAS describing the calling environment is pushed onto the process's ASH. Upon ASRETURN, the called environment is destroyed and the calling environment is restored by popping it from the ASH.

VM maintains two ASH registers, referring to the current user process ASH and the interrupt process ASH (IASH). (The justification for a separate interrupt ASH is discussed below.) VM provides operations to set these registers, corresponding to the action of a context swap. Each ASH is implemented in a separate segment (of type VM), so that a module responsible for scheduling, for instance, can own a collection of ASHs as components of process descriptors. Hence VM provides tho primitive mechanisms necessary to support the notion of processes, without overly specifying the nature of a process.

2.5. Virtual Interrupts

We have reasoned that putting a protected addressing environment mechanism at the lowest level of the system leads to strong modularity, and various other benefits. Such a philosophy would suggest that a hardware interrupt should be mapped into a protected procedure call, so that even interrupt processing routines can receive the benefits of the protection mechanism. However, we quickly discovered that efficiency precluded such a simple mechanism, because of the cost of the protected procedure call. Our first solution was to have VM "modify" all hardware devices into virtual devices, which had less stringent timing constraints. This design was unsatisfactory, because it was an instance of the philosophy, "for efficiency, put it in the kernel", which we sought to avoid. Our final solution was to put into VM a virtual interrupt *mechanism*, then let the low-level interrupt routines share the addressing environment in which VM resides, even though conceptually the device routines executed on top of the virtual machine defined by VM.

2.5.1 Devices as processes

Devices are modeled as low-level processes which orddnarily execute on peripheral hardware, but which sometimes call routines which must be executed on the CPU (interrupt routines). The device processes compete with the current user process for access to the CPU. This competition is arbitrated by the hardware priority mechanism. The user process may of course be executing in either user space or kernel space; the device routines execute in kernel space. Thus it is only programming convention which separates the device routines from the implementation of VM, even though conceptually they are in separate modules.

2.5.2 Virtual Interrupt Vectors

The process model for devices takes care of the worst of the timing constraints on devices; *it remains to provide a synchronization mechanism between the device* processes and the user system. For this purpose VM provides a set of *virtual interrupt vectors*, and operations for setting them and for evoking them. Each register is dedicated to a particular hardware device, and may be set to contain the name of an SAS and a function within that SAS which is to be executed when a virtual interrupt for that device occurs. Then any device process which needs to notify the user system of some event may signal the occurrence of a virtual interrupt from its device, via a VM operation.

Since the VM implementation is non-reentrant, virtual interrupts can only be processed when the user process is not executing a VM operation. Consequently virtual interrupts are entered in a system of priority queues, which is polled by VM after every VM operation. (Note that raising a virtual interrupt is a VM operation, so that if the user process is interrupted by a device process while in user space, the virtual interrupt is fielded as soon as it is raised.) Naturally, the queue of virtual interrupts is protected from simultaneous access by masking all interrupts while modifying it.

The mechanism just described permits the low level interrupt routines to be stacked by the *hardware mechanism in the usual way, so that classical real-time programming* techniques apply. Only certain critical sections of VM turn off all interrupts, namely the sections which save and restore state on interrupts and traps, and the virtual interrupt mechanism itself.

2.5.3 Virtual Interrupt Handling

When VM polls the virtual interrupt vectors after an operation, and discovers a pending interrupt, it mus force an ASCALL to the address space entry point listed in the vector. Instead of using the user's ASH for executing the protected procedures VM uses a separate address space history, the IASH, for two reasons:

1. For billing purposes, it will be preferable to separate DAS's associated with "the system" from those associated with a particular user.

2. Interrupts often precipitate rescheduling operations. The scheduler must be able to swap user contexts (by resetting the ASH register) and still be able to return to the active DAS's associated with its own and other virtual interrupt routines.

All virtual interrupts share the same IASH; a high priority routine may force a currently executing lower priority one to be stacked on the IASH, and later resumed when the former is completed.

2.6. Virtual Traps

The functional hierarchy of the modules of VM would not be practical for a real operating system unless some means were available for exception handling. For this purpose VM provides a set of virtual trap vectors, each associated with a particular exceptional condition which VM is not programmed to handle. When such a condition occurs, VM aborts whatever it was doing and forces an ascall to the function named in the corresponding virtual trap vector.In addition to conditions which arise within VM, there are a set of uninterpreted trap vectors which higher levels of the operating system may use for exception handling. For instance, when the clock module detects that one of its virtual clocks has run out of time, it cannot call the module to which the virtual clock belongs, because that would violate the functional hierarchy. Instead, the clock module associates a virtual trap vector with each virtual clock, and requires the user of a clock to set the corresponding trap vector with an appropriate function; then when the clock's value drops to zero, the clock module may evoke a virtual trap to whatever function is named in the trap vector.

3. Some Implemented Modules

3.1. Clocks

In order to accommodate family members with different clock usage requirements, two versions of the clock module were implemented. Both versions implement identical specifications, but vary in the relative speeds of clock operations and in their storage requirements. The clock module uses the hardware line clock to provide a collection of clocks, removing the line clock from the virtual machine presented to higher levels of the system. The clocks could be used by higher levels for scheduling, accounting or performance measurements.

There are seven operations applicable to a clock. A clock can be turned on and off and can have its alarm set or turned off ("enabling" and "disabling" the clock). The clock time can be read and set. The function to be called when a clock's alarm goes off can be specified. When a clock is running, its time is decremented once every time unit (0.1 seconds for the present system). If a clock is running with its alarm set and it decrements past zero, an interrupt is issued, calling the function associated with the clock. At this level of the system, there is no attempt to enforce the notion of ownership of a clock. A clock resource manager would be designed at a higher level of some family members.

One implementation of the clock module uses an array of clocks. Since it is too expensive to update clocks every time unit, all clocks contain their time as of the last clock interrupt. At that interrupt, the line clock was set to the shortest time value of the running, enabled clocks. This value is saved in LASTVAL. At any instant, the actual value of a running clock is: clocktime - (LASTVAL - lineclocktime). This implementation makes changes to the state of a clock quick and easy. In most cases, turning a clock on or off, disabling or enabling its alarm, or reading or setting a clock time requires changing the values in the clock, and perhaps changing LASTVAL and the line clock. However, handling interrupts requires that every clock in the array be checked to determine whether or not it is running and should be updated. All clocks must be inspected, even if only one is in use.

The second implementation of the clock module maintains all running clocks in a doubly-circularly-linked list. Each running clock contains the difference between its time and the time of the clock preceding it, rather than its actual time. The next clock to decrement past zero is designated FIRSTRUNNING, and the clock whose alarm goes off next is designated FIRSTENABLED. To obtain a running clock's actual time, add the values of all clocks between FIRSTRUNNING and that clock, and subtract (LASTVAL-line clock time) as in the array implementation. In the list implementation, changing the state of a clock is relatively expensive because inserting and removing clocks from the ring is complex. However, interrupts can be handled very quickly, since one need only scan down the list past any clocks which have gone off and reset the values of FIRSTRUNNING, FIRSTENABLED, LASTVAL and the line clock.

A system would use the array implementation if it expected to perform clock operations more frequently than clock interrupts occurred, or if space were tight, since the array implementation doesn't need space for storage links. If the number of running clocks varied greatly during the running of the system, the array implementation would be poor, since interrupt handling always checks the maximum number of clocks ever used. The list implementation would be preferable if the clock module would be primarily handling clock interrupts.

Since the specifications of both implementations are identical, they can be used interchangeably, depending on performance requirements.

3.2. Processes

The process module implements the concept of a *process.* A process is the sequential flow of control through address spaces; it is represented by an address space history. A process is executed by loading its ASH into the virtual stack register provided by VM. The process module is decomposed into several functional levels. The lowest level, *process management*, has been designed and implemented. Process creation is envisioned as a functional level depending on segment creation, and thus residing higher in the hierarchy.

Process management makes available a fixed number of processes and *sets* to which processes belong. Processes in process sets are maintained in their order of arrival and have waiting values associated with them. The *ready list*, a process set managed by the process module, represents a collection of processes that are ready to be executed on a processor. The currently running process is the first element on that list. A round-robin scheduling policy is implemented on the ready list with the aid of a virtual clock. A virtual trap at the end of a specified number of time intervals signals that the time slice which has been allotted to the currently running process has elapsed. This process is then moved to the end of the ready list and the next process is dispatched.

The process management level provides a collection of *waiting lists* (sets of processes) and operations to move processes between waiting lists and the ready list. A process always resides in exactly one process set. The current process can block by invoking the operation *haltme*, which removes the currently running process from the ready list and places it into a specified waiting list with a given waiting value. A process is reactivated through a *conditional continue* operation: A specified waiting list is scanned from the beginning for a process with a *waiting key* satisfying a condition. If there is at least one such process, it is transferred to the ready list. The waiting key of a process can be tested to see if it is equal to a value, not less than a value, or if the logical AND of the key and a value is non-zero. This mechanism is sufficiently general to allow for priority scheduling on waiting lists and for the implementation of several synchronization mechanisms.

3.3. Synchronization

Synchronization is an example of the generation of two family members based on the same underlying virtual machine. *Semaphores* and *path expressions* are provided as synchronization mechanisms. Either or both modules may exist in a system, depending upon which synchronization tools are required by higher levels of the system.

3.3.1 Semaphores

The semaphore module implements *counting semaphores* [Habermann 72] with P and V operations. A semaphore consists of a counter and a waiting list for blocked processes, provided by the process module. Processes waiting on a semaphore are reactivated in first-in first-out order using the operations on waiting lists provided by the process module.

3.3.2 Path Expressions

The goal of path expressions is to state the concurrency restrictions on a shared object at a higher level, analogous to control flow constructs like the *while*-statement. A shared object is described by a type definition, i.e. a specification of its data structure and a collection of operations for manipulation of an object of that type. A path expression, defined for an object as part of its type definition, describes the allowable sequences of operations, guaranteeing mutual exclusion of operations on the shared object. All information about the concurrency restrictions on a shared object is localized in the path expression for its data type.

The *basic path expression* is a regular expression from which all possible execution sequences can be derived. Its operands are the function names of operations, and the operators are *repetition* (*), *sequencing* (;), and *exclusive selection* (+) (in precedence order, which can be overruled by parentheses). The path expression is delimited by a *Path End* pair, which implies repetition of the whole path expression. For example the path expression for a file

path open ; (read * + write) ; close *end*
requires first the execution of an *open*, then either one *write* or (exclusively) zero or more *reads*, which must be followed by a *close* before the path expression can be repeated.

A path expression can be defined by a deterministic finite state machine, and can be represented by a directed graph in which the nodes correspond to states and an arc labeled with function name *p* indicates the execution of *p*. A function may execute in more than one path expression state if repetition of function names is permitted in a path expression.

Upon entry to a function a *prologue* is executed which determines whether the function is permitted to execute by checking the current path expression state associated with the shared object. The process either is blocked on the path expression waiting list or enters the function body. The waiting key of a blocked process contains all possible states in which it may start execution. At the end of a function execution, an *epilogue* performs the state transition of the path expression. If the path expression waiting list contains a process which may execute from the new state, the epilogue activates the process and releases the critical section on the shared object.

Several extensions have been considered to the basic path expression [Habermann 75]Campbell!thesis}. We have implemented a restricted form of the *numerical path element.* A numerical path element permits specification of additional constraints on the execution sequence of operations. It limits the number of invocations of two functions relative to one another. For example the path expression

path (push - pop) ^{n} *end*

restricts the number of *push* and *pop* operations on a bounded stack to satisfy: $\# pop \leq \# push \leq \# pop + n$. In order to permit a simple and efficient implementation of the numerical path element, a function name in a numerical path element may not appear more than once in a path expression. Every numerical path element in a path expression has a counter containing the difference in the number of invocations of its two functions and its own waiting list. A process trying to execute a function in a numerical path element will be blocked if the invocation count would not satisfy the numerical path element constraint. It can only be reactivated by another function in the same numerical path element. The process then proceeds to check the path expression state. The numerical path element condition is not required to be reevaluated if the process blocks on the path expression state.

4. Conclusion

Our first experience with the design of a system family is favorable. We are confident that we will be able to design several differing family members. Fundamental to our design are the notions of level and module. The levels are constructed via an incremental machine design. This method greatly enhances the design and debugging processes because it becomes possible to concentrate on one level at a time. The module concept leads to an unconventional ordering of the levels. Traditionally, one finds the multiprogramming and processor allocation facilities immediately above the hardware. However, since protection of modules is common to all family members whereas the processor allocation strategies may differ from one member to another, the level placed on top of the hardware is the one which implements the protected address spaces in which modules operate. Other levels which have been designed include a virtual clock level to reside immediately above VM, and the process definition level which resides above that.

The virtual memory system described has been implemented on a PDP-11/45 with segmentation feature. We argue that it is a virtual machine as we have defined, by providing a definition of the base machine and the modifications made to it by this first level.

The VM1 machine is the PDP-11/45 with

1) the program status word, relocation registers, segmentation status registers, register set 0, and emulate trap word hidden and therefore unavailable to the user. Also, the halt, wait, reset, and emulate instructions are no longer available.

2) new complex registers added, namely address spaces, working sets, known address space tables, the ASH, etc. Also new instructions, namely "segload", "segunload", "ascall", "asreturn", etc. are added to the instruction set.

3) all memory references by instructions systematically altered from 16 bit physical addresses to {working set slot, displacement} pairs. All interrupt and trap vectors are systematically altered from 16 bit physical addresses to {address space, FT index} pairs.

A module is described at various abstract levels so that its meaning does not have to be derived from the code. The building blocks for modules are type definitions. These allow us to separate specification from implementation issues. Type definitions provide yet another protection tool by limiting the extent of bugs. Continuation of the research effort will produce several running family members with highly non-trivial differences, including batch and timesharing systems with widely differing storage management strategies.

5. References

[Dijkstra 68] Dijkstra, E. W., The Structure of the T.H.E. Multiprogramming System. *Comm. ACM 11,5* (May 1968), 341-346.

[Flon 75] Flon, L., Program Design With Abstract Data Types. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. (June 1975).

[Habermann 72] Habermann, A. N., Synchronization of Communicating Processes. *Comm. ACM 15,3* (March 1972), 171-176.

[Habermann 73] Habermann, A. N., Integrated Design. *SIGPLAN Notices* (Sept. 1973), 64-66.

[Habermann 75] Habermann, A. N., Path Expressions. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. (June 1975).

[Liskov 72] Liskov, B., The Design of the VENUS Operating System. *Comm. ACM 15,3* (March 1972), 144-149.

[Neumann 74] Neumann, P. G. et al, On the Design of a Provably Secure Operating System. Proc. IRIA Workshop on Protection in Operating Systems, Paris (Aug. 1974), 161-175.

[Parnas 72a] Parnas, D. L. and Siewiorek, D. P., Use of the Concept of Transparency in the Design of Hierarchically Structured Systems. Department of Computer Science, Carnegie-Mellon University,Pittsburgh, Pa. (Sept. 1972).

[Parnas 72b] Parnas, D. L., On the Criteria to be Used in Decomposing Systems into Modules. *Comm. ACM 15,12* (Dec. 1972), 1053-1058.

[Parnas 74] Parnas, D. L., On a 'Buzzword': Hierarchical Structure. Proceedings of the IFIPS Congress 74, Stockholm, Sweden (1974).

[Price 73] Price, W. R., Implications of a Virtual Memory Mechanism for Implementing Protection in a Family of Operating Systems. Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University,Pittsburgh, Pa. (June 1973).